Finite Automata

Extra slide material (see whiteboard)

Why automata models?

- **Automaton**: Strongly limited computation model compared to ordinary computer programs
  - A weak model (with many limitations) ...
  - allows to do static analysis
    - e.g. on termination (decidable for finite automata)
    - which is not generally possible with a general computation model
  - is easy to implement in a general-purpose programming model
    - e.g. scanner generation/coding, parser generation/coding
    - source code generation from UML statecharts
  - Generally, we are interested in the weakest machine model (automaton model) that is still able to recognize a class of languages.

Finite Automaton / Finite State Machine

- Given by quintuple \((\Sigma, S, s_0, F, \delta)\)
  - \(\Sigma\): alphabet
  - \(S\): set of states
  - \(s_0\): initial state
  - \(F\): set of final states
  - \(\delta\): transition function

Transition table \(\delta\)

- Transitions in \(\delta\) are tuples \(((s, t), s')\)
  - \(s\): current state
  - \(t\): input symbol
  - \(s'\): new state

Computation of a Finite Automaton

- Initial configuration:
  - current state := start state \(s_0\)
  - read head points to first symbol of the input string
- 1 computation step:
  - read next input symbol, \(t\)
  - look up \(\delta\) for entry \((current\ state, t, new\ state)\)
  - current state := new state
  - move read head forward to next symbol on tape
- if all symbols consumed and new state is a final state: accept and halt
  - otherwise repeat

NFA and DFA

**NFA** (Nondeterministic Finite Automaton)
- "empty moves" (reading \(\epsilon\)) with state change are possible, i.e. entries \(((s, \epsilon, s')\) may exist in \(\delta\)
- ambiguous state transitions are possible, i.e. entries \((s, t, s)\) and \((s, t, s')\) may exist in \(\delta\)

**NFA accepts** input string if there exists a computation (i.e., a sequence of state transitions) that leads to "accept and halt"

**DFA** (Deterministic Finite Automaton)
- No \(\epsilon\)-transitions, no ambiguous transitions (\(\delta\) is a function)
- Special case of a NFA

DFA Example

- DFA with Alphabet \(\Sigma = \{0, 1\}\) initial state: \(s_0\)
  - State set \(S = \{s_0, s_1\}\)
  - \(F = \{s_1\}\)
  - \(\delta = \{(s_0, 0, s_0), (s_0, 1, s_1), (s_1, 0, s_0), (s_1, 1, s_0)\}\)

**recognizes (accepts)** strings containing an odd number of 1s

Computation for input string 10110:

- \(s_0\) read 1
- \(s_1\) read 0
- \(s_0\) read 1
- \(s_1\) read 1
- \(s_0\) accept
From regular expression to code

4 Steps:
- For each regular expression $r$ there exists a NFA that accepts $L_r$ [Thompson 1968 - see whiteboard]
- For each NFA there exists a DFA accepting the same language
- For each DFA there exists a minimal DFA (min. #states) that accepts the same language
- From a DFA, equivalent source code can be generated. [→Lecture on Scanners]

Theorem: For each regular expression $r$ there exists an NFA that accepts $L_r$ [Thompson 1968]

Proof: By induction, following the inductive construction of regular expressions

Divide-and-conquer strategy to construct NFA($r$):
1. If $r$ is trivial (base case): construct NFA($r$) directly, else:
2. Decompose $r$ into its constituent subexpressions $r_1$, $r_2$...
3. Recursively construct NFA($r_1$), NFA($r_2$), ...
4. Compose these to NFA($r$) according to decomposition of $r$

2 base cases:
Case 1: $r = \varepsilon$
- NFA($r$) =
- recognizes $L(\varepsilon) = \{\varepsilon\}$

Case 2: $r = a$ for $a \in \Sigma$
- NFA($r$) =
- recognizes $L(a) = \{a\}$

4 recursive decomposition cases:

Case 3: $r = r_1 \mid r_2$
- By Ind.-hyp. exist NFA($r_1$), NFA($r_2$)
- NFA($r$) =
- recognizes $L(r_1 \mid r_2) = L(r_1) \cup L(r_2)$

Case 4: $r = r_1 \cdot r_2$
- By Ind.-hyp. exist NFA($r_1$), NFA($r_2$)
- NFA($r$) =
- recognizes $L(r_1 \cdot r_2) = L(r_1) \cdot L(r_2)$

Case 5: $r = r_1^*$
- By Ind.-hyp. exists NFA($r_1$)
- NFA($r$) =
- recognizes $L(r_1^*) = (L(r_1))^*$

Case 6: Parentheses: $r = (r_1)$
- NFA($r$) =
- (no modifications)

Theorem follows by induction.